Neural, Parallel & Scientific Computations 15 (2007) 1-26

A Highly Efficient Implementation of Back Propagation Algorithm using Matrix Instruction Set Architecture

Mostafa I. Soliman and Samir A. Mohamed
Computer & Control Section, Electrical Engineering Department,
Faculty of Engineering, South Valley University, Aswan, Arab Republic of Egypt

Abstract
Back Propagation (BP) training algorithm has received intensive research efforts to exploit its parallelism in order to reduce the training time for complex problems. A modified version of BP based on matrix-matrix multiplication was proposed for parallel processing. This paper discusses the implementation of Matrix Back Propagation (MBP) using scalar, vector, and matrix instruction set architecture (ISA). Besides, it shows that the performance of the MBP is improved by switching form scalar to vector ISA and form vector to matrix ISA. On a practical application, speech recognition, the speedup of training a neural network using unrolling scalar over scalar ISA is 1.83. On eight parallel lanes, the speedup of using vector, unrolling vector, and matrix ISA are respectively 10.33, 11.88, and 15.36, where the maximum theoretical speedup is 16. Our results show that the use of matrix ISA gives a performance close to the optimal because of reusing the loaded data, decreasing the loop overhead, and overlapping the memory operations by arithmetic operations.

Keywords – neural networks, parallel architecture, vector/matrix processing, parallel algorithms, back propagation algorithm, reusing data, loop unrolling

1. INTRODUCTION

A major challenge facing computer designers is determining how to improve performance by executing multiple operations in parallel. These parallel operations can be defined by programmers, extracted statically by compliers, and/or extracted dynamically by hardware. If the extracted parallelism depends only on the skills of the programmer, compiler, or hardware, improving performance requires a professional programmer,

Received August 1, 2005; final form December 11, 2007

sophisticated complier, or complex hardware. For example, superscalar architectures [1, 2] dynamically extract parallelism from a single instruction stream to execute multiple independent scalar instructions in the same clock cycle. Thus, the die area used to actually execute parallel operations is relatively small with respect to the area needed for extracting parallel operations. Besides, the hardware complexity grows at least quadratically with issue width [3]. To treat the problem of increasing the hardware complexity, a compiler should help hardware for extracting parallelism. VLIW architectures [4] figure out this problem by packing parallel scalar instructions during the compilation time into a very long instruction. Very sophisticated compiler techniques are required to detect and schedule a peak amount of instruction parallelism.

Scalar instruction set architectures (ISAs), which are used in superscalar and VLIW architectures, require a separate opcode and related operand specifiers for every operation to be performed. Using scalar ISAs increase the semantic gap between high-level languages and hardware [5]. Moreover, the parallelism convoyed by programmers through high-level statements using high-level languages is lost when a scalar ISA is used. However, vector architectures [6-13], which have a two-level ISA, provide vector instructions (1-D scalar operations) to support 1-D data parallelism in addition to scalar instructions (0-D data parallelism) to support unvectorizable codes. Vector-vector, vector-scalar, and scalar-scalar instructions are used for coding an application on vector computers. Nowadays, single chip vector processors, which have a four-way superscalar core for processing unvectorizable data and parallel execution datapaths (eight parallel lanes) for processing vector data, are already fabricated [14].

The logical extension of vector processing is matrix processing, where the parallel execution datapaths can be used not only for processing vector data but also for processing matrix data. Our proposed Trident processor [15-17], which has a three-level ISA, provides a scalar core for processing scalar data and a matrix engine for processing matrix/vector data. The Trident architecture extends the advantages of the vector ISA by adding a matrix instruction set. The semantic content of the Trident matrix and vector instructions already includes the notion of parallel operations.

In this paper, the advantages of using high-level ISA (such as matrix and vector ISAs) over using low-level ISA (such as scalar ISA) are demonstrated. Thus, an application based on a mixture of scalar, vector, and matrix operations is needed. Error Back Propagation (BP) training algorithm is a well studied and famous algorithm in the artificial neural network area. However, the implementation of BP algorithm on a scalar processor has several drawbacks. The most important one is the training time it takes for complex problems (networks having lots of neurons and/or for problems having a large number of training and validation patterns). One obvious solution to alleviate this problem is to use other efficient training algorithms. For example, Levenberg-Marquardt

[18] can speedup the training process significantly, however, it has a memory requirement that reduces its feasibility on complex problems. Another solution for the training time problem is the use of parallel processing to reduce the training time of BP.

The BP algorithm is based on a mixture of scalar, vector-vector (level-1 BLAS [19]), and matrix-vector operations (level-2 BLAS [20]). This means that the performance of the BP algorithm using scalar ISA can be dominated by the amount of memory traffic rather than by the number of arithmetic operations involved. The ratio of arithmetic operations to data movements would be improved to further avoid excessive data movements to and from the main memory because the movement of data can be as costly as (or even higher than) arithmetic operations on the data. This problem can be reduces by reusing the loaded data. The modified version of the back propagation algorithm, which is called matrix back propagation (MBP) is a good choice. MBP is based on a mixture of scalar, vector-vector (level-1 BLAS), matrix-vector (level-2 BLAS), and matrix-matrix operations (level-3 BLAS [21-23]). Mainly, it is based on matrix-matrix multiplication, which is the operation of choice for high performance computing. To show the advantage of using a higher level ISA, the number of clock cycles needed for the MBP is calculated in case of using scalar, vector, and matrix ISA. Besides, the speedup of using vector and matrix ISAs over using scalar ISA on MBP is calculated.

The remainder of this paper is organized as follows. In Section 2, the MBP learning algorithm is described. It shows the derivation of MBP from the well known BP algorithm and demonstrates the conversion of the MBP algorithm into matrix form. Section 3 depicts the architecture of the Trident processor and illustrates the Trident implementation of the MBP algorithm. Performance evaluation of the MBP algorithm using scalar, vector, and matrix ISA is demonstrated and discussed in Section 4. Finally, Section 5 presents our conclusions and future directions of work.

2. MATRIX BACK PROPAGATION ALGORITHM

In this section, an overview of the BP training algorithm is given as it is the basis of the rest of this paper. In addition, the mathematical formulation of the MBP algorithm is provided as given in [24, 25].

The objective of BP is to train neural networks having any number of hidden units arranged in any number of layers. The only restriction is that the connection pattern between layers must not contain cycles. This kind of networks is called *feedforward* networks. The BP algorithm consists of two phases: the forward and backward phases. In the forward phase, the activations are propagated from the input layer to the output layer through all the hidden layers, as shown in Figure 1. While in the backward phase, the error between the observed actual outputs and the desired output values in the output and

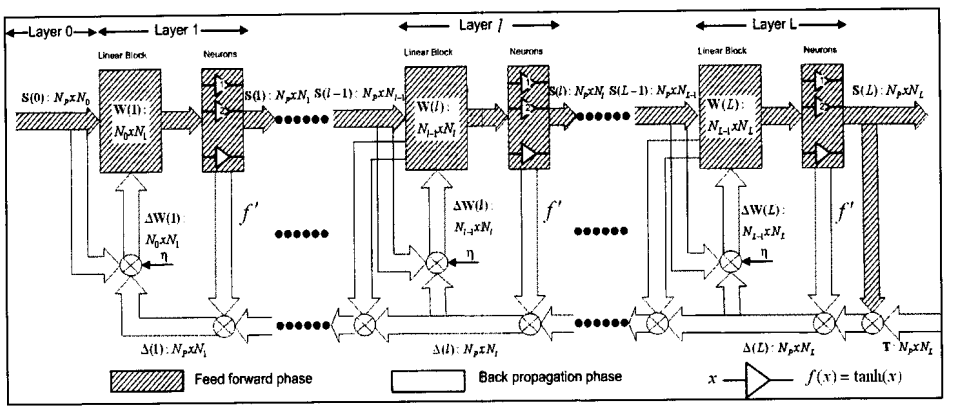


Figure 1: The block diagram of the MBP training algorithm

bias values to reduce the error. To train a multi-layer feedforward network, the gradient descent algorithm is used to approximate an unknown function, based on some training data, $S(0)$, and the corresponding target (desired output), T. The overall gradient with respect to the entire training set is the sum of the gradients for each pattern.

As depicted in Figure 1, the input of the network is indicated as layer 0; it contains no real neurons because its purpose is to spread the input to the neurons of the first hidden layer. The hidden layers are numbered from 1 to L-1. The output layer is L. In general, the l^{th} layer contains N_l neurons. Thus, the input layer has N_0 elements and the output layer has N_L neurons. A neuron n in layer l is connected to all the neurons in layer l-1 through N_{l-1} connections; each one is associated to a weight. These weights is organized in a vector $w^{(n)}(l)$. Besides, the corresponding bias will be $b_n(l)$.

Assuming that the training data consists of N_P patterns $\{s^{(p)} \mid p=1...N_P\}$, applying a pattern $s^{(p)}$ to the input layer, it propagates from input to output through every hidden layer. Each layer responds with a precise pattern that is called the *status* of that layer. In general, the l^{th} layer will be in status $s^{(p)}(l)$ and the output of the network will be $s^{(p)}(L)$. The status of the n^{th} neuron of the l^{th} layer is computed with the feed-forward rule:

$$s_n^{(p)}(l) = f\left(s^{(p)}(l-1)^T . w^{(n)}(l) + b_n(l)\right) = f\left(\sum_{i=1}^{N_{l-1}} s_i^{(p)}(l-1) * w_i^{(n)}(l) + b_n(l)\right)$$

where f is the activation function of the neuron. Hyperbolic tangent (*tanh*) is used as an activation function in this paper. The total error E is defined as the normalized squared difference between the output vector of the network when the pattern vector $s(0)^{(p)}$ is applied and the corresponding desired output vector $t^{(p)}$, where $p \in \{1, 2, \ldots, N_P\}$.

$$E = \frac{1}{N_P N_L} \sum_{p=1}^{N_P} \left\| t^{(p)} - s^{(p)}(L) \right\|^2 = \frac{1}{N_P N_L} \sum_{p=1}^{N_P} \sum_{n=1}^{N_L} \left(t_n^{(p)} - s_n^{(p)}(L) \right)^2$$

A simple method to minimize E with respect to the weights is to start from a random point in the weight space and then descend step-by-step towards a minimum of E. If the minimum is satisfactory the algorithm stops, otherwise another random point is chosen and the descent is repeated. In order to choose the right direction, at each step the gradient of E (∇E) is computed. The gradient finds the direction of maximum growth of E, because of that the right direction is $-\nabla E$. The gradient descent algorithm is summarized in the following steps:

1) Start with random weights $w^{(n)}(l)$; weights of neuron n in layer l.
2) Compute $\nabla E = \frac{\partial E}{\partial w_i^{(n)}(l)}$ for each weight.
3) Compute the step in the opposite direction ($\Delta w_i^{(n)}(l) = -\eta \frac{\partial E}{\partial w_i^{(n)}(l)}$), where η is the learning constant.
4) Update the weights ($w_i^{(n)}(l) = w_i^{(n)}(l) + \Delta w_i^{(n)}(l)$).
5) Calculate E

   If E satisfies the predefined error constraint then stop

   Else if E does not satisfy the predefined error constraint and the algorithm is stuck in a local minimum then go to step 1

   Else go to step 2.

To compute the gradient (one of the most expensive steps in the BP algorithm), the derivatives in the output layer is computed as follows:

$$\frac{\partial E}{\partial w_i^{(n)}(L)} = \frac{\partial\left[\frac{1}{N_P N_L}\sum_{p=1}^{N_P}\sum_{j=1}^{N_L}\left(t_j^{(p)} - s_j^{(p)}(L)\right)^2\right]}{\partial w_i^{(n)}(L)} = \frac{-2}{N_P N_L}\sum_{p=1}^{N_P}\left[\left(t_j^{(p)} - s_j^{(p)}(L)\right)\frac{\partial s_n^{(p)}(L)}{\partial w_i^{(n)}(L)}\right]$$

It is known that $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$.

$$\frac{\partial E}{\partial w_i^{(n)}(L)} = \frac{-2}{N_P N_L}\sum_{p=1}^{N_P}\left[\left(t_n^{(p)} - s_n^{(p)}(L)\right) * \left(1 - \left[s_n^{(p)}(L)\right]^2\right) * s_i^{(p)}(L-1)\right]$$

Thus, the above equation can be written in a more compact form as follows:

$$\frac{\partial E}{\partial w_i^{(n)}(L)} = -\sum_{p=1}^{N_P}\delta_n^{(p)}(L) * s_i^{(p)}(L-1), \text{ where,}$$

$$\delta_n^{(p)}(L) = \frac{-2}{N_P N_L}\left(t_n^{(p)} - s_n^{(p)}(L)\right) * \left(1 - \left[s_n^{(p)}(L)\right]^2\right)$$

The update rule for the weights in the output layer is the famous delta-rule:

$$\Delta w_i^{(n)}(L) = \eta \sum_{p=1}^{N_P}\delta_n^{(p)}(L) * s_i^{(p)}(L-1)$$

For the weights in the remaining layers, the procedure is the same as layer *L-1* as bellow:

$$\frac{\partial E}{\partial w_i^{(n)}(L-1)} = \frac{\partial}{\partial w_i^{(n)}(L-1)}\left[\frac{1}{N_P N_L}\sum_{p=1}^{N_P}\sum_{j=1}^{N_L}\left(t_j^{(p)} - s_j^{(p)}(L)\right)^2\right]$$

To get rid of the deep embedding of the weights in layer *L*-1 the chain rule is used for the derivative (indicating with $E^{(p)}$ the partial error due to pattern *p*):

$$\frac{\partial E}{\partial w_i^{(n)}(L-1)} = \frac{1}{N_P}\sum_{p=1}^{N_P}\left[\frac{\partial E^{(p)}}{\partial s_n^{(p)}(L-1)} * \frac{\partial s_n^{(p)}(L-1)}{\partial w_i^{(n)}(L-1)}\right] =$$

$$\frac{1}{N_P}\sum_{p=1}^{N_P}\left[\frac{\partial E^{(p)}}{\partial s_n^{(p)}(L-1)} * \left(1-\left[s_n^{(p)}(L-1)\right]^2\right) * s_i^{(p)}(L-2)\right] = -\sum_{p=1}^{N_P}\delta_n^{(p)}(L-1) * s_i^{(p)}(L-2)$$

where, $\delta_n^{(p)}(L-1) = \left(1-\left[s_n^{(p)}(L-1)\right]^2\right) * \sum_{j=1}^{N_L}\left[\delta_j^{(p)}(L) * w_n^{(j)}(L)\right]$

The back-propagation algorithm can be summarized in the following steps.

1) **Feed-forward**

for each layer l := 1 to L
  for each neuron n := 1 to N_l
    for each pattern p := 1 to N_P

$$s_n^{(p)}(l) = f\left(s^{(p)}(l-1)^T . w^{(n)}(l) + b_n(l)\right)$$

2) **Error computation and error back-propagation**

for each neuron in the output layer n := 1 to N_L
  for each pattern p := 1 to N_P

$$\delta_n^{(p)}(L) = \frac{2}{N_P N_L}\left(t_n^{(p)} - s_n^{(p)}(L)\right)\left(1-\left[s_n^{(p)}(L)\right]^2\right)$$

for each layer l := L-1 to 1
  for each neuron n := 1 to N_l
    for each pattern p := 1 to N_P

$$\delta_n^{(p)}(l) = \left(1-\left[s_n^{(p)}(l)\right]^2\right) * \sum_{j=1}^{N_L}\left[\delta_j^{(p)}(l+1) * w_n^{(j)}(l+1)\right]$$

3) **Step computation**

for each layer l := 1 to L
  for each neuron n := 1 to N_l

$$\Delta b_n(l) = \eta\sum_{p=1}^{N_P}\delta_n^{(p)}(l)$$

  for each weight i := 1 to N_{l-1}

$$\Delta w_i^{(n)}(l) = \eta\sum_{p=1}^{N_P}\delta_n^{(p)}(l) * s_i^{(p)}(l-1)$$

4) **Weight updating**

for each layer l := 1 to L
  for each neuron n := 1 to N_l

$$b_n^{new}(l) = b_n^{old}(l) + \Delta b_n(l)$$

  for each weight i := 1 to N_{l-1}

$$w_i^{(n),new}(l) = w_i^{(n),old}(l) + \Delta w_i^{(n)}(l)$$

The above described BP training algorithm can be converted into matrix form, which is called Matrix Back Propagation (MBP) training algorithm (see [24, 25] for more details and see Figure 1 for an illustrative block diagram). The status of neurons in layer l can be easily organized in matrices $S(l)$ of size $N_P \times N_l$, where N_P is the number of input patterns and N_l is the number of neurons in layer l:

$$S(l) = \begin{bmatrix} s_1(l) & s_2(l) & \cdots & s_{N_l}(l) \end{bmatrix} = \begin{bmatrix} s^{(1)T}(l) \\ s^{(2)T}(l) \\ \vdots \\ s^{(N_P)T}(l) \end{bmatrix} = \begin{bmatrix} s_1^{(1)}(l) & s_2^{(1)}(l) & \cdots & s_{N_l}^{(1)}(l) \\ s_1^{(2)}(l) & s_2^{(2)}(l) & \vdots & s_{N_l}^{(2)}(l) \\ \vdots & \vdots & \ddots & \vdots \\ s_1^{(N_P)}(l) & s_2^{(N_P)}(l) & \cdots & s_{N_l}^{(N_P)}(l) \end{bmatrix}$$

The row p of this matrix contains the status of all the neurons in layer l when a pattern p is applied to the network. As a special case, the input patterns are organized in the same way in matrix $S(0)$.

Similarly, the weights can be arranged in matrices $W(l)$ of size $N_{l-1} \times N_l$:

$$W(l) = \begin{bmatrix} w^{(1)}(l) & w^{(2)}(l) & \cdots & w^{(N_l)}(l) \end{bmatrix} = \begin{bmatrix} w_1^T(l) \\ w_2^T(l) \\ \vdots \\ w_{N_{l-1}}^T(l) \end{bmatrix} = \begin{bmatrix} w_1^{(1)}(l) & w_1^{(2)}(l) & \cdots & w_1^{(N_l)}(l) \\ w_2^{(1)}(l) & w_2^{(2)}(l) & \vdots & w_2^{(N_l)}(l) \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_{l-1}}^{(1)}(l) & w_{N_{l-1}}^{(2)}(l) & \cdots & w_{N_{l-1}}^{(N_l)}(l) \end{bmatrix}$$

The column n of matrix $W(l)$ contains the weights of neuron n of layer l. The propagated errors also can be arranged in matrices $\Delta(l)$ of size $N_P \times N_l$:

$$\Delta(l) = \begin{bmatrix} \delta_1(l) & \delta_2(l) & \cdots & \delta_{N_l}(l) \end{bmatrix} = \begin{bmatrix} \delta^{(1)T}(l) \\ \delta^{(2)T}(l) \\ \vdots \\ \delta^{(N_P)T}(l) \end{bmatrix} = \begin{bmatrix} \delta_1^{(1)}(l) & \delta_2^{(1)}(l) & \cdots & \delta_{N_l}^{(1)}(l) \\ \delta_1^{(2)}(l) & \delta_2^{(2)}(l) & \vdots & \delta_{N_l}^{(2)}(l) \\ \vdots & \vdots & \ddots & \vdots \\ \delta_1^{(N_P)}(l) & \delta_2^{(N_P)}(l) & \cdots & \delta_{N_l}^{(N_P)}(l) \end{bmatrix}$$

The row p of matrix $\Delta(l)$ contains the error propagated to layer l when pattern p is applied. In addition, a matrix T of size $N_P \times N_L$ with the same structure of $S(L)$ is needed to store the target patterns. The variation of weights and biases computed at each step are stored respectively in matrices $\Delta W(l)$ of size $N_{l-1} \times N_l$ and vectors $\Delta b(l)$ of length N_l.

Now the BP algorithm can be rewritten in matrix form as shown in Table 1. The steps (1a), (2c), and (3a) are marked to emphasize that the computational load belongs to them is $O(n^3)$ complexity, where n is the size of the problem.

Table 1: The MBP algorithm

Step	Description	FLOPs	Operation	High Level Operation
Feed-forward				
(1a)	$S(l) = S(l\text{-}1) \times W(l)$	$2N_PN_lN_{l-1}$	*, +	Matrix-matrix multiplication
(1b)	$S(l) = S(l) + b(l) \cdot 1^t$	N_PN_l	+	Matrix-vector addition
(1c)	$S(l) = f\{S(l)\}$	$k_1N_PN_l$	e^x, e^{-x}, +, -, /	Element-wise matrix operation
Error back-prop				
(2a)	$\Delta(L) = T - S(L)$	N_PN_L	-	Matrix-matrix subtraction
(2b)	$\Delta(L) = \Delta(L) * g\{S(L)\}$	$(k_2+1)N_PN_L$	*, *, -	Element-wise matrix operation
(2c)	$\Delta(l) = \Delta(l+1) \times W^t(l+1)$	$2N_PN_lN_{l-1}$	*, +	Matrix-matrix multiplication
(2d)	$\Delta(l) = \Delta(l) * g\{S(l)\}$	$(k_2+1)N_PN_l$	*, *, -	Element-wise matrix operation
Weight variation				
(3a)	$\Delta W(l)^{new} = S^t(l\text{-}1) \times \Delta(l)$	$2N_PN_lN_{l-1}$	*, +	Matrix-matrix multiplication
(3b)	$\Delta b(l)^{new} = \Delta^t(l) \cdot 1$	N_PN_l	+	Vector accumulation
(3c)	$\Delta W(l)^{new} = \eta\Delta W(l)^{new} + \alpha\Delta W(l)^{old}$	$3N_lN_{l-1}$	*, *, +	Element-wise matrix operation
(3d)	$\Delta b(l)^{new} = \eta\Delta b(l)^{new} + \alpha\Delta b(l)^{old}$	$3N_l$	*, *, +	Element-wise vector operation
Weigh update				
(4a)	$W(l) = W(l) + \Delta W(l)^{new}$	N_lN_{l-1}	+	Element-wise matrix addition
(4b)	$b(l) = b(l) + \Delta b(l)^{new}$	N_l	+	Element-wise vector addition

FLOPs: floating-point operations
$f(x)$: tanh(x)
α: the influence of the previous steps on the current one
k_1: the number of operations needed to compute $f()$

3. TRIDENT IMPLEMENTATION OF MBP

3.1 Trident Microarchitecture

Figure 2 shows an overall block diagram of the Trident processor [15, 17]. Scalar, vector, and matrix instructions of an application are fetched from the instruction cache and stored in the fetch buffer awaiting sending to the proper unit. The dispatch unit splits the incoming instruction stream into scalar instructions and high-level (vector/matrix) instructions. Scalar instructions are executed on the scalar core, which includes traditional scalar functional units and a scalar register file.

The scalar core in the Trident processor can be in-order/out-of-order, single/multiple-issue. Practically, the vector processor developed for NEC SX-6 supercomputers integrates a four-way out-of-order superscalar processor with eight vector lanes on a single chip (60 million transistors) [14]. The Trident scalar unit is responsible for executing scalar (unparallel) code and for supporting high-level vector/matrix instructions. In other words, the primary job of the scalar core is to serve the matrix unit to do bookkeeping computation and to do any code that cannot be done effectively using high-level vector/matrix instructions. Again, the scalar core is not responsible for achieving high performance.

Our proposed extended unit (matrix unit) for processing vector/matrix data is based on the decoupled technique [26]. A decoupled processor has two independent units, the address unit and the computation unit. The address unit performs all address computations and loads/stores data from/to main memory to/from register files. The

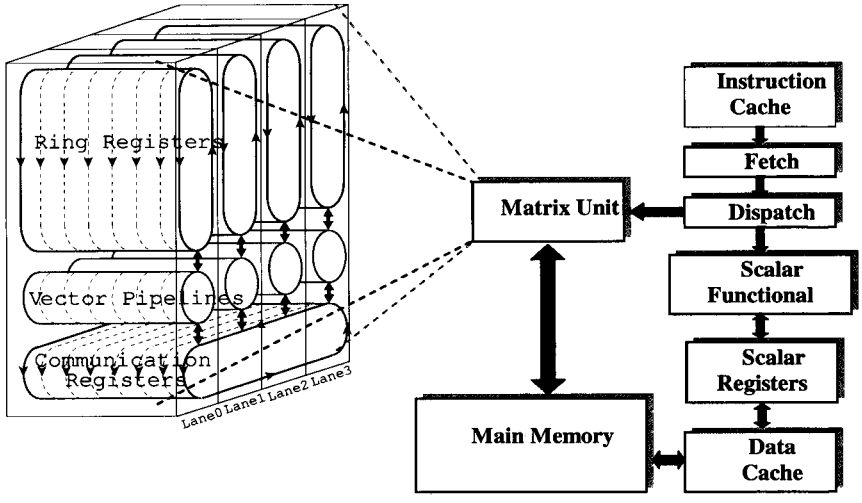


Figure 2: Trident processor block diagram

computation unit executes all arithmetic instructions on data loaded into registers. These units are communicated through architectural queues which are used to temporary keep the loaded/stored data from/to the main memory to/from the register file.

The Trident matrix unit has two types of register file based on shift registers needed for vector/matrix processing. A ring register file had been proposed for storing and cyclically shifting 1-D data within a lane. A communication register file had been proposed for storing and cyclically shifting 1-D data across parallel lanes. By using ring and communication register files, data stored in parallel lanes can be cyclically shifted in 2-D space (horizontally and vertically), which is needed for vector/matrix processing.

As shown in Figure 2, Trident microarchitecture is based on local communications because it is known that processors are rapidly becoming bound by wire delay [27-29]. Besides, the organization of the Trident processor is based on parallel lanes, which reduces the design and verification complicity. Each lane contains a set of ring registers and functional units. P ring register (one ring per lane) represents a matrix register, which can store as well as cyclically shift (within lanes) vector/matrix data.

3.2 Trident Implementation of Matrix-matrix Multiplication

It is clear from Table 1 that the computational load belongs to steps (1a), (2c), and (3a) that show $O(n^3)$ complexity, where n is the size of the problem. To compute these steps, three matrix-matrix multiplications must be performed: step (1a) is a conventional matrix product ($C = A \times B$), step (2c) is a matrix product with the second matrix transposed ($C = A \times B^t$), and step (3a) is a matrix product with the first matrix transposed ($C = A^t \times B$). Most of the remaining steps if Table 1 are based on element-wise matrix operations (steps (1c), (2a), (2b), (2d), (3c), and (4a) that show $O(n^2)$ complexity) and element-wise vector

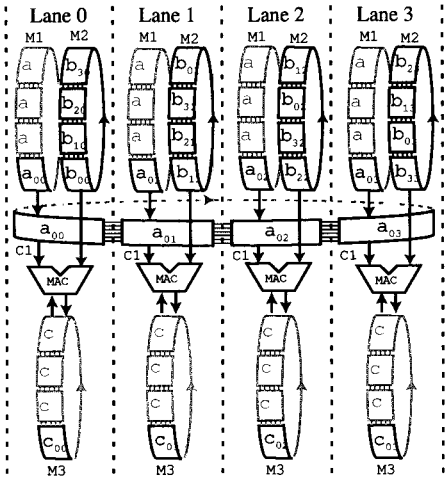


Figure 3: Trident Implementation of the conventional matrix product ($C = A \times B$)

operations (steps (3d) and (4b) that show $O(n)$ complexity). Step (1b) can be considered as a rank-one update operation (the second vector is constant) or matrix-vector addition. Finally, step (3b) is based on accumulating vectors to scalar values.

Figure 3 shows the implementation of the conventional matrix-matrix multiplication ($C = A \times B$) on four Trident lanes. A block (4×4 elements) of matrix A is loaded into a matrix register (see the content of the M1 matrix register, which has four rings; each has four elements). Only four memory addresses are needed for loading this block since A is an aligned matrix. By the same way, a block of matrix B is loaded into a matrix register (see the content of the M2 matrix register in Figure 3). After loading M2, skewing operation should be done to adjust the elements of B for parallel processing, as shown in Figure 4. It is easy to show that skewing a block of a $P \times P$ block requires $P/2$ clock cycles, since the ring registers can be cyclically shifted in both directions.

The processing time for multiplying two $P \times P$ blocks of matrices on P Trident lanes is P^2 clock cycles. However, the loading time is $2P$ clock cycles, assuming P elements can be loaded per clock cycle. For $P \geq 4$, the loading and skewing times ($2P + P/2$ clock cycles) can be overlapped by the processing time (P^2 clock cycles). Table A1 in the appendix illustrates in detail the execution of the conventional matrix-matrix multiplication on four ($P = 4$) Trident lanes. It shows the input of each lane every clock

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

(a) Before skewing

a_{00}	a_{11}	a_{22}	a_{33}
a_{10}	a_{21}	a_{32}	a_{03}
a_{20}	a_{31}	a_{02}	a_{13}
a_{30}	a_{01}	a_{12}	a_{23}

(b) After skewing

Figure 4: Skewing the input matrix for matrix-matrix multiplication

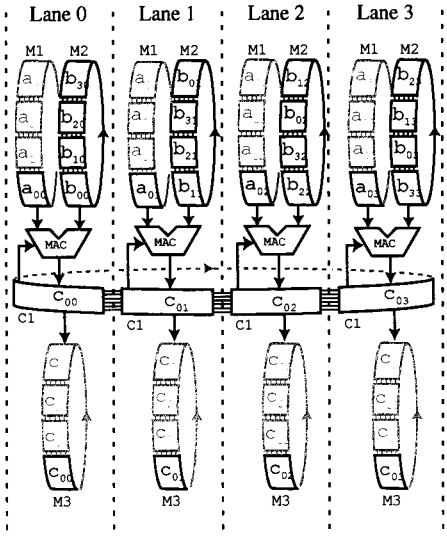


Figure 5: Trident Implementation of a matrix product $C = A \times B^t$

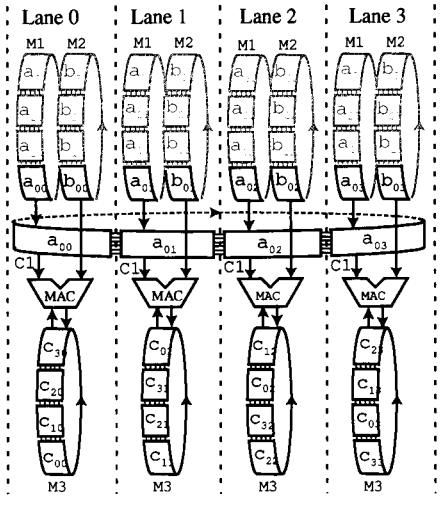


Figure 6: Trident Implementation of a matrix product $C = A^t \times B$

cycle and the results after P clock cycles.

Figure 5 shows the Trident Implementation of a matrix product with the second matrix transposed. It is based on vector-matrix multiplication as a middle loop and dot-product as an inner loop. Like conventional matrix product, a block (4×4 elements) of matrix A is loaded into a matrix register (M1) and a block of matrix B is loaded and skewed into a matrix register (M2). After P^2 (=16) clock cycles the results of matrix-matrix multiplication will be available in the matrix register M3 (see Table A2 for more detail). Similarly, Figure 6 shows the Trident Implementation of a matrix product with the first matrix transposed. In this case, the middle loop is based on outer-product and the inner loop is based on SAXPY (a scalar times vector X plus vector Y).

The key advantage of the Trident implementation of $C = A \times B$, $C = A \times B^t$, and $C = A^t \times B$ is the use of unit stride (contiguous accessing) for loading/storing matrices. The input matrices will be loaded into matrix registers row by row and the output results will be stored also row by row (assuming the matrices are stored in the memory row major). Other key advantages of the Trident implementation of matrix products are reusing the loaded data and overlapping both the loading/storing times and loop overheads by computation time.

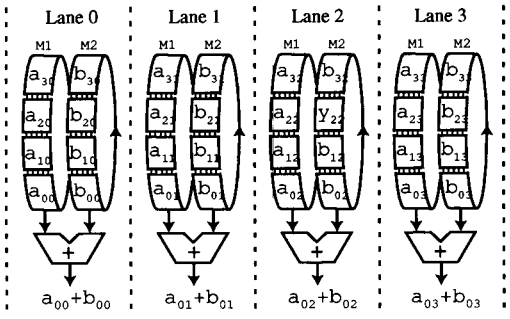


Figure 7: Trident Implementation of the element-wise matrix addition ($C=A+B$)

3.3 Trident Implementation of Element-wise Matrix/Vector Operations

Element-wise matrix-matrix instructions ($C=A$ op B), such as matrix addition, subtraction, multiplication, division, etc., are executed on the Trident processor without cross-lane communications. Figure 7 shows the implementation of element-wise matrix-matrix addition on a four-lane Trident processor. A 4×4 block of each of the input matrices A and B are loaded into M1 and M2 matrix registers. An execution datapath within a lane can process data stored in ring registers at the rate of one element per clock cycle, as shown in Table A4. Each datapath receives identical control but different input elements in each clock cycle by cyclically shifting the ring registers in parallel.

Element-wise vector instructions ($z = x$ op y), such as vector addition, subtraction, multiplication, division, etc., can be processed on multiple execution datapaths similar to element-wise matrix-matrix instructions. Figure 8 shows the implementation of vector addition on a four-lane Trident processor. The input x and y vectors are distributed across a set of ring registers in a round-robin fashion (see the content of M1 and M2 matrix registers in Figure 8). An execution datapath (vector pipeline) within a lane can process vector data stored in ring registers at the rate of one element per clock cycle..

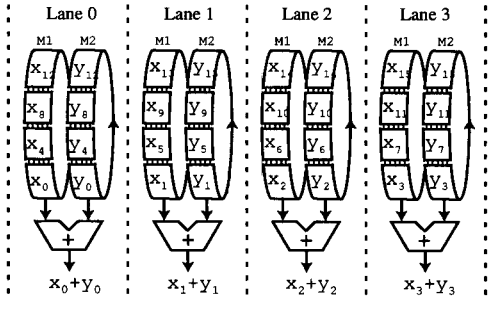


Figure 8: Trident Implementation of the element-wise vector addition ($z=x+y$)

4. PERFORMANCE EVALUATION OF MBP

In this section, the performance of the MBP algorithm is evaluated when scalar, vector, and matrix ISA are used. Firstly, the performance of MBP is evaluated on a hypothetic neural network with the following parameters. The number of layers is four ($L = 4$). Three sizes of the training set (N_P) are selected: 100, 1000, and 10,000. The number of neurons in a layer l (N_l) is assumed to be equal to that in layer l-1 ($N_0 = N_1 = N_2 = N_3 = N_4$). Then, the performance of the MBP is evaluated on a practical problem, speech recognition, with two layers ($N_0 = 234$, $N_1 = 1000$, and $N_2 = 69$).

4.1 Metric

Time is the measure of computer performance; obviously, the computer that performs the same amount of work in the least time is the fastest. The execution time of a computer program can be expressed roughly as the product of three terms: the number of instructions required, the average number of clock cycles per instruction, and the time per clock cycle [30]. Since the goal is to maximize performance (minimize the execution time), a computer designer wants to decrease each of these terms. Unfortunately, these factors are interrelated, where the number of instructions executed in a program depends on the instruction set architecture (scalar, vector, and matrix ISA), and the other two factors depend on the architecture organization and the choice of implementation technology, as well as the ISA [31].

The use of high-level ISA, such as matrix ISA, reduces the number of instructions needed for coding a program. Besides, the clock cycle time can be drastically reduced on the Trident processor because of using local interconnections within and between parallel lanes. The remaining factor, the average number of clock cycles per instruction, decreases on the Trident because of reducing the overhead due to looping.

4.1 Performance Evaluation of the MBP using scalar ISA

The number of clock cycles needed for MBP using scalar ISA can be expressed as:

$$\#\text{ Cycles} = 6N_P\sum_{l=1}^{L}N_lN_{l-1} - 2N_PN_1N_0 + (6+k_1+k_2)N_P\sum_{l=1}^{L}N_l + 6\sum_{l=1}^{L}N_lN_{l-1} + N_PN_L + 6\sum_{l=1}^{L}N_l$$

The multiply-accumulate operation is assumed to be performed in a single clock cycle. Figure 9 shows the calculation of the number of clock cycles in more details. Moreover, the number of floating-point operations (FLOPs) in the MBP algorithm (see Table 1) can be calculated as follows.

$$\#\text{ FLOPs} = 6N_P\sum_{l=1}^{L}N_lN_{l-1} - 2N_PN_1N_0 + (3+k_1+k_2)N_P\sum_{l=1}^{L}N_l + 4\sum_{l=1}^{L}N_lN_{l-1} + N_PN_L + 4\sum_{l=1}^{L}N_l$$

$6N_P(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4) - 2N_0N_1 +$	Steps (1a), (2c), (3a)
$8N_P(N_1 + N_2 + N_3 + N_4) +$	Steps (1b), (1c), (3b)
$4N_PN_4 +$	Steps (2a), (2b)
$3N_P(N_1 + N_2 + N_3) +$	Steps (2d)
$4(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4) +$	Steps (3c), (4a)
$4(N_1 + N_2 + N_3 + N_4) +$	Steps (3d), (4b)
$(N_1 + N_2 + N_3 + N_4) +$	Storing b(1), b(2), b(3), b(4)
$(N_1 + N_2 + N_3 + N_4) +$	Storing Δb(1), Δb(2), Δb(3), Δb(4)
$N_P(N_1 + N_2 + N_3 + N_4) +$	Storing S(1), S(2), S(3), S(4)
$N_P(N_1 + N_2 + N_3 + N_4) +$	Storing Δ(1), Δ(2), Δ(3), Δ(4)
$(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4) +$	Storing W(1), W(2), W(3), W(4)
$(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4)$	Storing ΔW(1), ΔW(2), ΔW(3), ΔW(4)

Figure 9: Number of clock cycles for MBP ($L = 4$) using scalar ISA

Thus, the number of clock cycles per FLOP (CPF) using scalar ISA is approximately one even though the multiply-accumulate operation is assumed to be performed in one clock cycle. In few words, $O(n^3)$ clock cycles are needed to perform $O(n^3)$ FLOPs on MBP algorithm using scalar ISA, where $N_P = N_l = N_{l-1} = n$.

On the other hand, the number of memory operations (load/store) using scalar ISA is $O(n^3)$. This means that the performance of the MBP algorithm using scalar ISA can be dominated by the amount of memory traffic rather than by the number of floating-point operations involved. The ratio of FLOPs to data movements would be improved to further avoid excessive data movements to and from the main memory because the movement of data can be as costly as (or even higher than) arithmetic operations on the data. To address this problem, the data would be loaded into registers or first level cache once and used many times to improve CPF.

In the above analysis, an important factor has not been considered, which is the overhead due to looping (or loop overhead). The number of ALU operations (updating the indices, comparing and branching back to the beginning) needed for looping is approximately as follows.

$$\#\text{ Operations for loop overhead} \approx 6N_P\sum_{l=1}^{L} N_lN_{l-1} - 2N_PN_1N_0 + O(N^2)$$

Implementing MBP by scalar ISA requires one extra ALU operation per FLOP, which is an unacceptable overhead. This means, the looping represents a source of overhead that should be decreased to improve the performance of an application.

To decrease the effect of loop overhead as well as to improve percentage of reusing the loaded data, loop unrolling technique [32] is used. The increase of the unrolling depth decreases the number of iterations per loop and shifts the balance of the loop from memory-bound to CPU-bound. Therefore, extra cycles are available for the memory port to load/store the operands while the processor is computing the arithmetic operations. In

other words, the loading/storing time can be overlapped by the execution of arithmetic operations. However, increasing the unrolling depth results in increasing the code size, which needs a larger instruction cache to reduce the cache miss rate.

Table 2 shows the scalar loop unrolling needed for matrix products in MBP algorithm (steps (1a), (2c), (3a) in Table 1, which represent the computational load in the algorithm). It is know that the unrolling depth is limited by the number of registers available for storing intermediate results. Assuming that n scalar registers are available for storing the intermediate scalar results, the parameters U and V can be easily calculated from $2U+1=n$ and $V^2+2V=n$, respectively (see Table 2). For n =32, the depth of loop unrolling U and V are respectively 15 and 4.

The number of clock cycles needed for MBP when using the scalar loop unrolling technique is:

$$(3+\frac{1}{U}+\frac{1}{V}+\frac{1}{V^2})N_P\sum_{l=1}^{L}N_lN_{l-1}-(1+\frac{1}{V})N_PN_1N_0+(6+k_1+k_2)N_P\sum_{l=1}^{L}N_l+6\sum_{l=1}^{L}N_lN_{l-1}+N_PN_L+6$$

The speedup due to using scalar unrolling over scalar code is shown in Figure 10, where the maximum speedup is approximately $6/(3 + 1/U + 1/V + 1/V^2)$. In case of $U = 15$ and $V = 4$, the maximum speedup is approximately 1.78. To show another advantage of the use of loop unrolling, the overhead due to looping should be considered.

$$\#\text{ Operations for loop overhead} \approx 2(\frac{1}{U}+\frac{2}{V^2})N_P\sum_{l=1}^{L}N_lN_{l-1}-\frac{2}{V^2}N_PN_1N_0+O(N^2)$$

Thus, the effect of loop overhead is decreased roughly by a factor equals to the depth of loop unrolling.

Table 2: Scalar loop unrolling of the matrix products in MBP

$S(l) = S(l\text{-}1) \times W(l)$	$\Delta(l) = \Delta(l+1) \times W^t(l+1)$	$\Delta W(l) = S^t(l\text{-}1) \times \Delta(l)$
for $j = 0$ to N_l-1 step U	for $j = 0$ to N_l-1 step V	for $i = 0$ to N_{l-1}-1 step V
for $i = 0$ to N_P-1	for $i = 0$ to N_P-1	for $j = 0$ to N_l-1 step V
for $k = 0$ to N_{l-1}-1	for $k = 0$ to N_{l+1}-1 step V	for $k = 0$ to N_P-1
$s^l_{i,j} += s^{l-1}_{i,k} * w^l_{k,j}$	$\delta^l_{i,j} += \delta^{l+1}_{i,k} * w^{l+1}_{j,k}$	$\Delta w^l_{i,j} += s^{l-1}_{k,i} * \delta^l_{k,j}$
$s^l_{i,j+1} += s^{l-1}_{i,k} * w^l_{k,j+1}$	$\delta^l_{i,j} += \delta^{l+1}_{i,k+1} * w^{l+1}_{j,k+1}$	$\Delta w^l_{i,j+1} += s^{l-1}_{k,i} * \delta^l_{k,j+1}$
:	:	:
$s^l_{i,j+U-1} += s^{l-1}_{i,k} * w^l_{k,j+U-1}$	$\delta^l_{i,j} += \delta^{l+1}_{i,k+V-1} * w^{l+1}_{j,k+V-1}$	$\Delta w^l_{i,j+V-1} += s^{l-1}_{k,i} * \delta^l_{k,j+V-1}$
	$\delta^l_{i,j+1} += \delta^{l+1}_{i,k} * w^{l+1}_{j+1,k}$	$\Delta w^l_{i+1,j} += s^{l-1}_{k,i+1} * \delta^l_{k,j}$
	:	:
	$\delta^l_{\ldots} += \delta^{l+1}_{\ldots} * w^{l+1}_{\ldots}$	$\Delta w^l_{\ldots} += s^{l-1}_{\ldots} * \delta^l_{\ldots}$

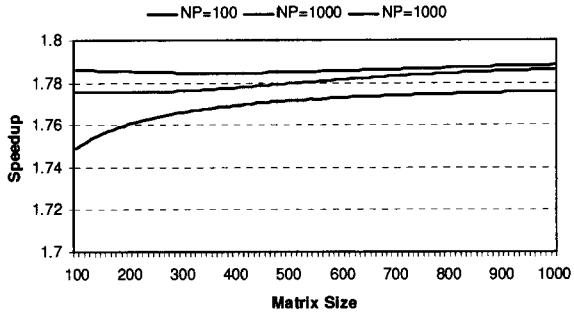


Figure 10: Speedup of scalar unrolling over scalar code (NP is the number of input patterns)

4.2 Performance Evaluation of the MBP using Vector ISA

The performance of the MBP can be further improved by using vector ISA. On P vector lanes, P parallel operations can be performed per clock cycle, which accelerate the execution of vector operations. Table 3 shows the vector pseudo code needed for the computational expensive parts in the MBP algorithm (steps (1a), (2c), (3a) in Table 1). The number of clock cycles using vector ISA is:

$$(\frac{4}{P}+\frac{2}{VL})N_P\sum_{l=1}^{L}N_lN_{l-1}-\frac{2N_PN_1N_0}{P}+\frac{(6+k_1+k_2)}{P}N_P\sum_{l=1}^{L}N_l+PN_P\sum_{l=2}^{L}N_l+\frac{6}{P}\sum_{l=1}^{L}N_lN_{l-1}+\frac{N_PN_L}{P}+\frac{6}{P}\sum_{l=1}^{L}N_l$$

where P is the number of parallel lanes and VL is the maximum length of a vector register. Figure 11 shows detailed calculation of the number of clock cycles needed each step in MBP when vector ISA is used on P parallel lanes. The speedup of using vector ISA over scalar ISA is shown in Figure 12, where the number of parallel lanes equals four and the optimal speedup is eight. There is another factor of speedup was not considered in Figure 11: reducing the loop overhead.

$$\#\text{ Operations for loop overhead} \approx \frac{6N_P}{VL}\sum_{l=1}^{L}N_lN_{l-1}-\frac{2}{VL}N_PN_1N_0+O(N^2)$$

Thus the loop overhead is reduced by a factor equals to the vector length, which represents a key advantage of using vector processing over scalar processing.

To further improve the performance of MBP using vector ISA, the unrolling technique can be considered (see Table 4). The number of clock cycles can be expressed as follows.

Table 3: Vector pseudo code for the computational expensive parts in the MBP algorithm

$S(l) = S(l\text{-}1) \times W(l)$	$\Delta(l) = \Delta(l+1) \times W^t(l+1)$	$\Delta W(l) = S^t(l\text{-}1) \times \Delta(l)$
for $j = 0$ to N_l-1 step VL for $i = 0$ to N_P-1 for $k = 0$ to N_{l-1}-1 $s^l_{i,j:j+VL-1} += s^{l-1}_{i,k} * w^l_{k,j:j+VL-1}$	for $i = 0$ to N_P-1 for $j = 0$ to N_l-1 for $k = 0$ to N_{l+1}-1 step VL $\delta^l_{i,j} += \delta^{l+1}_{i,k:k+VL-1} * w^{l+1}_{j,k:k+VL-1}$	for $j = 0$ to N_l-1 step VL for $i = 0$ to N_{l-1}-1 for $k = 0$ to N_P-1 $\Delta w^l_{i,j:j+VL-1} += s^{l-1}_{k,i} * \delta^l_{k,j:j+VL-1}$

$$\text{\# Cycles} = \begin{aligned} &(\frac{3}{P}+\frac{1}{PV^2}+\frac{2}{UVL})N_P\sum_{l=1}^{L}N_lN_{l-1}-(\frac{1}{P}+\frac{1}{PV^2})N_PN_1N_0 \\ &+\frac{(6+k_1+k_2)}{P}N_P\sum_{l=1}^{L}N_l+PN_P\sum_{l=2}^{L}N_l+\frac{6}{P}\sum_{l=1}^{L}N_lN_{l-1}+\frac{N_PN_L}{P}+\frac{6}{P}\sum_{l=1}^{L}N_l \end{aligned}$$

Not only the number of clock cycles is further decreased using unrolling technique but loop overhead also is decreased.

$$\text{\# Operations for loop overhead} \approx 2N_P(\frac{2}{UVL}+\frac{1}{V^2VL})\sum_{l=1}^{L}N_lN_{l-1}-\frac{2}{UVL}N_PN_1N_0+O(N^2)$$

The speedup of using both vector ISA and loop unrolling technique is greater than using vector ISA only as shown in Figure 13. The maximum speedup is approximately 5.7, which represents 84% of the optimal speedup.

$2(1/P + 1/VL)N_P(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4)$ +	Steps (1a), (3a)
$2N_P/P(N_1N_2 + N_2N_3 + N_3N_4) + PN_P(N_2 + N_3 + N_4)$ +	Steps (2c)
$8N_P/P(N_1 + N_2 + N_3 + N_4)$ +	Steps (1b), (1c), (3b)
$4N_PN_4/P$ +	Steps (2a), (2b)
$3N_P/P(N_1 + N_2 + N_3)$ +	Steps (2d)
$4(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4)$ +	Steps (3c), (4a)
$4/P(N_1 + N_2 + N_3 + N_4)$ +	Steps (3d), (4b)
$1/P(N_1 + N_2 + N_3 + N_4)$ +	Storing b(1), b(2), b(3), b(4)
$1/P(N_1 + N_2 + N_3 + N_4)$ +	Storing Δb(1), Δb(2), Δb(3), Δb(4)
$N_P/P(N_1 + N_2 + N_3 + N_4)$ +	Storing S(1), S(2), S(3), S(4)
$N_P/P(N_1 + N_2 + N_3 + N_4)$ +	Storing Δ(1), Δ(2), Δ(3), Δ(4)
$1/P(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4)$ +	Storing W(1), W(2), W(3), W(4)
$1/P(N_0N_1 + N_1N_2 + N_2N_3 + N_3N_4)$	Storing ΔW(1), ΔW(2), ΔW(3), ΔW(4)

Figure 11: Number of clock cycles for MBP ($L = 4$) using vector ISA

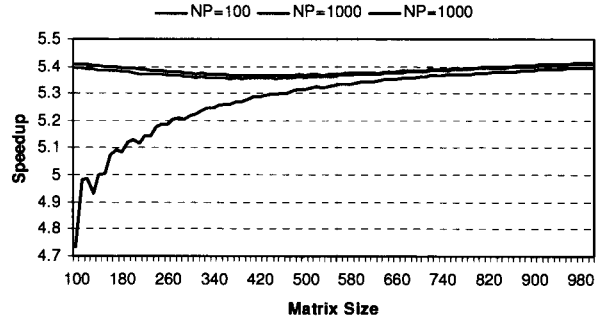


Figure 12: Speedup of vector over scalar code (NP is the number of input patterns)

Table 4: Vector loop unrolling of the matrix products in MBP

$S(l) = S(l\text{-}1) \times W(l)$	$\Delta(l) = \Delta(l+1) \times W^t(l+1)$	$\Delta W(l) = S^t(l\text{-}1) \times \Delta(l)$
for $j = 0$ to N_l-1 step VL for $i = 0$ to N_P-1 step U for $k = 0$ to N_{l-1}-1 $s^l_{i,j:j+VL-1} += s^{l-1}_{i,k} * w^l_{k,j:j+VL-1}$ $s^l_{i+1,j:j+VL-1} += s^{l-1}_{i+1,k} * w^l_{k,j:j+VL-1}$ ⋮ $s^l_{i+U-1,j:j+VL-1} += s^{l-1}_{i+U-1,k} * w^l_{k,j:j+VL-1}$	for $i = 0$ to N_P-1 Step V for $j = 0$ to N_l-1 Step V for $k = 0$ to N_{i+1}-1 step VL $\delta^l_{i,j} += \delta^{l+1}_{i,k:k+VL-1} * w^{l+1}_{j,k:k+VL-1}$ $\delta^l_{i,j+1} += \delta^{l+1}_{i,k:k+VL-1} * w^{l+1}_{j+1,k:k+VL-1}$ ⋮ $\delta^l_{i+V-1,j+V-1} += \delta^{l+1}_{i+V-1,k:k+VL-1} * w^{l+1}_{j+V-1,k:k+VL-1}$	for $j = 0$ to N_l-1 step VL for $i = 0$ to N_{l-1}-1 step U for $k = 0$ to N_P-1 $\Delta w^l_{i,j:j+VL-1} += s^{l-1}_{k,i} * \delta^l_{k,j:j+VL-1}$ $\Delta w^l_{i+1,j:j+VL-1} += s^{l-1}_{k,i+1} * \delta^l_{k,j:j+VL-1}$ ⋮ $\Delta w^l_{i+U-1,j:j+VL-1} += s^{l-1}_{k,i+U-1} * \delta^l_{k,j:j+VL-1}$

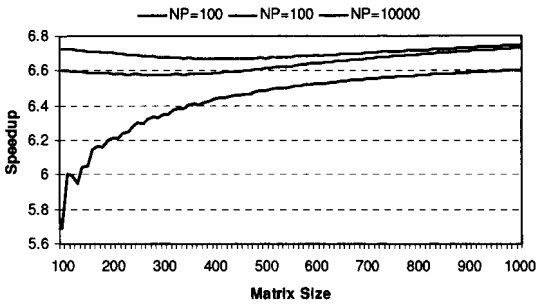


Figure 13: Speedup of vector and loop unrolling over scalar code (NP is the number of input patterns)

4.3 Performance Evaluation of the MBP using Matrix ISA

The logical next step is the use of matrix ISA to improve the performance of MBP. The block mining technique (see Table 5) is used to divide the input matrices into block instead of using the strip mining technique in vector processing. The processing of blocks by matrix ISA deceases the total number of clock cycles because of the highly reusing the loaded data [33]. Besides, the overhead due to looping can be neglected compared to the corresponding values of using vector and scalar ISAs.

$$\#\text{ Cycles} = \frac{3N_P}{P}\sum_{l=1}^{L} N_l N_{l-1} - \frac{N_P N_1 N_0}{P} + \frac{(6+k_1+k_2)}{P} N_P \sum_{l=1}^{L} N_l + \frac{6}{P}\sum_{l=1}^{L} N_l N_{l-1} + \frac{N_P N_L}{P} + \frac{6}{P}\sum_{l=1}^{L} N_l$$

$$\#\text{ Operations for loop overhead} \approx \frac{2N_P}{P^3}\sum_{l=1}^{L} N_l N_{l-1} - \frac{2}{P^3} N_P N_1 N_0 + O(N^2)$$

The speedup of using matrix ISA over scalar ISA is very close to the optimal value, as shown in Figure 14. The optimal value of the speedup is eight since the number of parallel lanes is four and two operation can be chained in a lane. The main reason for good performance of MBP algorithm using matrix ISA is overlapping the loading/storing time by the computation time.

Table 5: Pseudo matrix instructions for the matrix products in MBP

$S(l) = S(l\text{-}1) \times W(l)$	$\Delta(l) = \Delta(l+1) \times W^t(l+1)$	$\Delta W(l) = S^t(l\text{-}1) \times \Delta(l)$
for $i = 0$ to N_P-1 step P for $j = 0$ to N_l-1 step P for $k = 0$ to N_{l-1}-1 step P $s^{l}_{i:i+P-1,j:j+P-1} +=$ $s^{l-1}_{i:i+P-1,k:k+P-1} * w^{l}_{k:k+P-1,j:j+P-1}$	for $i = 0$ to N_P-1 step P for $j = 0$ to N_l-1 step P for $k = 0$ to N_{l-1}-1 step P $\delta^{l}_{i:i+P-1,j:j+P-1} +=$ $\delta^{l+1}_{i:i+P-1,k:k+P-1} * w^{l+1}_{j:j+P-1,k:k+P-1}$	for $i = 0$ to N_P-1 step P for $j = 0$ to N_l-1 step P for $k = 0$ to N_{l-1}-1 step P $\Delta w^{l}_{i:i+P-1,j:j+P-1} +=$ $s^{l-1}_{k:k+P-1,i:i+P-1} * \delta^{l}_{k:k+P-1,j:j+P-1}$

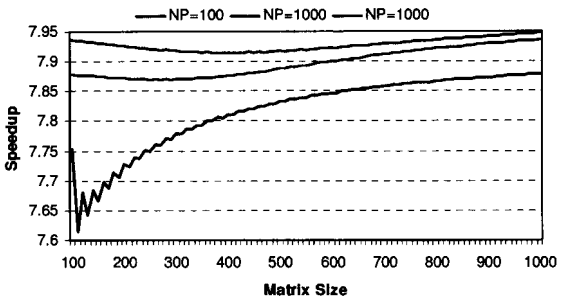


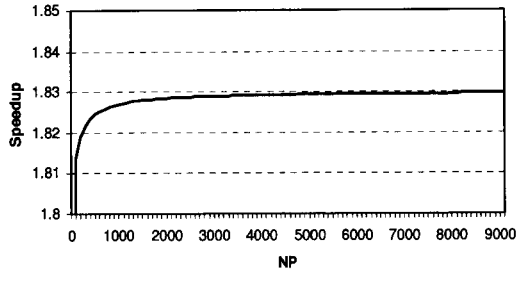
Figure 14: Speedup of using matrix ISA over scalar ISA (NP is the number of input patterns)

4.4 Practical Problem: Speech Recognition

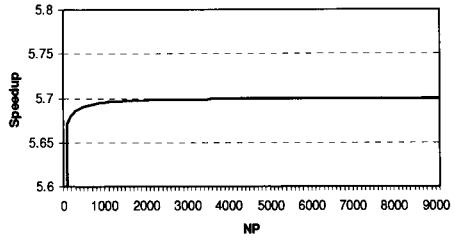
In this subsection, the performance of the MBP algorithm for training a neural network on speech recognition (a real problem) is shown using scalar, vector, and matrix ISA. This speech recognition neural network has only one hidden layer. The input layer has 234 neurons (N_0 = 234). The hidden layer has 1000 neurons (N_1 = 1000), and the output layer has 69 neurons (N_2 = 69). The number of input patterns for training the neural network (N_P) varies from 0 to 10000

The maximum speedup due to scalar loop unrolling is 1.83 (see Figure 15a). Moreover, on four parallel lanes, the maximum speedup of using vector and matrix ISA are 5.7 and 7.77, respectively, as shown in Figures 15b and 15d. These speedup represent 71% and 97% of the optimal value, which is eight. The performance of using vector ISA can be improved from 71% to 87% of the optimal value by using the loop unrolling technique, as shown in Figure 15c.

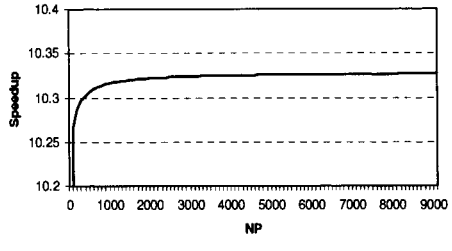
On eight parallel lanes, the maximum speedup of using vector, unrolling vector, and matrix ISA are 10.33, 11.88, and 15.36, respectively, as shown in Figures 15e, 15f, and 15g. To show the advantage of using matrix ISA over vector ISA, the speedup of using vector ISA represents 65% of the optimal value on eight lanes, however, the speedup of using matrix ISA represents 96% of the optimal value. This analysis shows the scalability of using matrix ISA



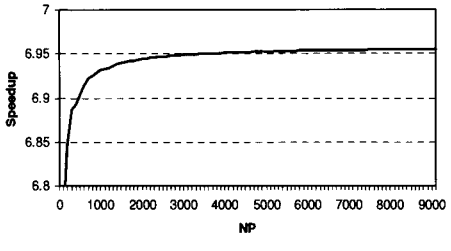
(a) Scalar ISA and loop unrolling



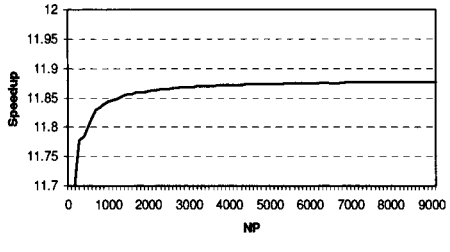
(b) Vector ISA ($P = 4$ and $VL = 16$)



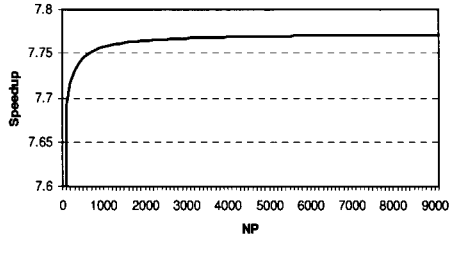
(e) Vector ISA ($P = 8$ and $VL = 64$)



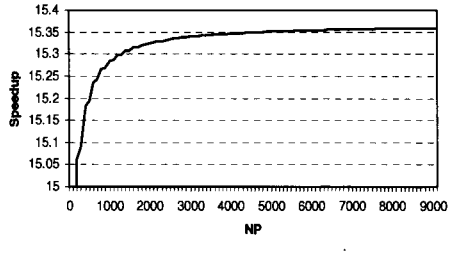
(c) Unrolling vector ISA ($P = 4$ and $VL = 16$)



(f) Unrolling vector ISA ($P = 8$ and $VL = 64$)



(d) Matrix ISA ($P = 4$)



(g) Matrix ISA ($P = 8$)

Figure 15: Speedup of using matrix/vector ISA over scalar ISA on MBP (NP is the number of input patterns)

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a highly efficient implementation of the Matrix Back Propagation (MBP) training algorithm for neural networks to solve complex problems requiring too much training time. Our implementation of MBP is based on the use of matrix ISA on a recently proposed matrix processor named Trident processor. Therefore, we briefly presented the architecture of this processor and its features. All the elementary matrix operations required for executing MBP and the number of clock cycles required are shown. We then evaluated the performance of the MBP algorithm when scalar, vector, and matrix ISA are used on a hypothetic neural network and on a practical problem (speech recognition). Our obtained results showed that unlike scalar and vector ISAs even by using loop unrolling, the speedup of the matrix ISA (on the Trident processor) can be very close to the optimal value. This is because of three reasons. First, the processing of blocks by matrix ISA deceases the total number of clock cycles due to the high reusing of the loaded data. Second, the overhead due to looping can be neglected compared to the corresponding values of using vector and scalar ISAs. Finally, the loading/storing time is overlapped by the computation time.

There are several possible future directions based on this work. By following the same approach shown in this paper, other practical problems can be evaluated and implemented on the Trident processor. Examples of these problems are parallel implementation of the LM algorithm, all-pair shortest path, and sales man problem.

REFERENCES

[1] J. Mike, *Superscalar Microprocessor Design*, Prentice Hall (Prentice Hall Series in Innovative Technology), 1991.

[2] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, Vol. 83, No. 12, December 1995, pp. 1609-24.

[3] S. Palacharla, *Complexity-Effective Superscalar Processors*, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1998.

[4] J. Fisher, "VLIW Architectures and the ELI-512," *Proc. 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pp 140-150.

[5] J. Smith, "The Best Way to Achieve Vector-Like Performance?," *Proc. 21st International Symposium on Computer Architecture*, Denver, CO, June 1997.

[6] O. Lubeck, J. Moore, and R. Mendez, "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *IEEE Computer*, December 1985.

[7] W. Schonauer, *Scientific Computing on Vector Computers*, North-Holland, Amsterdam, 1987.

[8] M. Awaga and H. Takahashi, "The μVP 64-bit Vector Coprocessor: A New Implementation of High-Performance Numerical Computation," *IEEE MICRO*, Vol. 3, No. 5, October 1993, pp. 24-36.

[9] C. Lee, *Code Optimizers and Register Organizations for Vector Architectures*, Ph.D Thesis, Computer Science Division, University of California at Berkeley, 1992.

[10] R. Espasa, *Advanced Vector Architectures*, Ph.D. Thesis, Department of Computer Architecture, Universitat Politecnica de Catalunya, Barcelona, Spain, February 1997.

[11] K. Asanovic, *Vector Microprocessors*, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1998.

[12] C. Kozyrakis, *A Media-enhanced Vector Architecture for Embedded Memory Systems*, Master Thesis, Computer Science Division, University of California at Berkeley, 1999.

[13] C. Kozyrakis, *Scalable Vector Media-processors for Embedded Systems*, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 2002.

[14] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh, "A Hardware Overview of SX-6 and SX-7 Supercomputer," *NEC Research & Development*, Vol. 44, No. 1, January 2003, pp. 2-7.

[15] M. Soliman and S. Sedukhin, "Technology Scalable Matrix Architecture for Data Parallel Applications," *IEICE Transactions on Information and Systems*, Vol. E86-D, No. 9, September 2003, pp. 1549-1559.

[16] M. Soliman and S. Sedukhin, "Matrix Bidiagonalization: Implementation and Evaluation on the Trident Processor," *Neural, Parallel & Scientific Computations*, Vol. 11, No. 4, December 2003, pp. 395-422.

[17] M. Soliman, *A Technology-Scalable Matrix Processor for Data Parallel Applications*, Ph.D. Thesis, Computer Science and Engineering, The University of Aizu, 2004.

[18] N. Ampazis and S. J. Perantonis. "Levenberg-marquardt Algorithm with Adaptive Momentum for Efficient Training for Feed Forward Networks," *Proc. International Joint Conference on Neural Networks - IJCNN*, Italy, 2000.

[19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, September 1979, pp. 308-323.

[20] J. Dongarra, J. Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, March 1988, pp. 1-17.

[21] J. Dongarra, J. Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, March 1990, pp. 1-17.

[22] J. Dongarra, I. Duff, D. Sorenson, and H. Van Der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, Society for Industrial and Applied Mathematics, 1991.

[23] J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, L. Torczon, and W. Gropp, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, November 2002.

[24] Davide Anguita, "MBP: A Matrix Back Propagation V1.1: An Efficient implementation of the BP algorithm," *Technical Report*, DIBE, University of Genova, Italy, 1993.

[25] Davide Anguita and B. Gomes, "MBP on T0: mixing floating- and fixed-point formats in BP learning," *Technical Report No. TR-94-038*, International Computer Science Institute, Berkeley, 1994.

[26] J. Smith, "Decoupled Access/Execute Computer Srchitectures," *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November 1984, pp. 289-308.

[27] J. Davis, R. Venkatesan, A. Kaloyeros, M. Beylansky, S. Souri, K. Banerjee, K. Saraswat, A. Rahman, R. Reif, and J. Meindl, "Interconnect Limits on Gigascale Integration (GSI) in the 21st Century," *Proceedings of the IEEE*, Vol. 89, No. 3, March 2001, pp. 305-324.

[28] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, Vol. 89, No. 4, April 2001, pp. 490-504.

[29] R. Ho, K. Mai, and M. Horowitz, "Efficient On-Chip Global Interconnects," *Proc. IEEE Symposium on VLSI Circuits*, June 2003, pp. 271- 274.

[30] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 3rd Edition, 2003.

[31] J. Hennessy and N. Jouppi, "Computer Technology and Architecture: An Evolving Interaction," *IEEE Computer*, Vol. 24, No. 9, September 1991, pp. 18-29.

[32] L. Song and K. Kavi, "What Can We Gain by Unfolding Loops?," *ACM SIGPLAN Notices*, Vol. 39, No. 2, February 2004, pp. 26-33.

[33] G. Golub and C. Van Loan, *Matrix Computations*, 3rd Edition, The Johns Hopkins University Press, Baltimore and London, 1996.

Appendix

Table A1: Trident execution of the conventional matrix product ($C = A \times B$)

Input	MAC0	MAC1	MAC2	MAC3
Clock Cycle 0	a_{00}, b_{00}	a_{01}, b_{11}	a_{02}, b_{22}	a_{03}, b_{33}
Clock Cycle 1	a_{01}, b_{10}	a_{02}, b_{21}	a_{03}, b_{32}	a_{00}, b_{03}
Clock Cycle 2	a_{02}, b_{20}	a_{03}, b_{31}	a_{00}, b_{02}	a_{01}, b_{13}
Clock Cycle 3	a_{03}, b_{30}	a_{00}, b_{01}	a_{01}, b_{12}	a_{02}, b_{23}
Results	$c_{00} = a_{00}*b_{00}+a_{01}*b_{10}+a_{02}*b_{20}+a_{03}*b_{30}$	$c_{01} = a_{01}*b_{11}+a_{02}*b_{21}+a_{03}*b_{31}+a_{00}*b_{01}$	$c_{02} = a_{02}*b_{22}+a_{03}*b_{32}+a_{00}*b_{02}+a_{01}*b_{12}$	$c_{03} = a_{03}*b_{33}+a_{00}*b_{03}+a_{01}*b_{13}+a_{02}*b_{23}$
Clock Cycle 4	a_{10}, b_{00}	a_{11}, b_{11}	a_{12}, b_{22}	a_{13}, b_{33}
Clock Cycle 5	a_{11}, b_{10}	a_{12}, b_{21}	a_{13}, b_{32}	a_{10}, b_{03}
Clock Cycle 6	a_{12}, b_{20}	a_{13}, b_{31}	a_{10}, b_{02}	a_{11}, b_{13}
Clock Cycle 7	a_{13}, b_{30}	a_{10}, b_{01}	a_{11}, b_{12}	a_{12}, b_{23}
Results	$c_{10} = a_{10}*b_{00}+a_{11}*b_{10}+a_{12}*b_{20}+a_{13}*b_{30}$	$c_{11} = a_{11}*b_{11}+a_{12}*b_{21}+a_{13}*b_{31}+a_{10}*b_{01}$	$c_{12} = a_{12}*b_{22}+a_{13}*b_{32}+a_{10}*b_{02}+a_{11}*b_{12}$	$c_{13} = a_{13}*b_{33}+a_{10}*b_{03}+a_{11}*b_{13}+a_{12}*b_{23}$
Clock Cycle 8	a_{20}, b_{00}	a_{21}, b_{11}	a_{22}, b_{22}	a_{23}, b_{33}
Clock Cycle 9	a_{21}, b_{10}	a_{22}, b_{21}	a_{23}, b_{32}	a_{20}, b_{03}
Clock Cycle 10	a_{22}, b_{20}	a_{23}, b_{31}	a_{20}, b_{02}	a_{21}, b_{13}
Clock Cycle 11	a_{23}, b_{30}	a_{20}, b_{01}	a_{21}, b_{12}	a_{22}, b_{23}
Results	$c_{20} = a_{20}*b_{00}+a_{21}*b_{10}+a_{22}*b_{20}+a_{23}*b_{30}$	$c_{21} = a_{21}*b_{11}+a_{22}*b_{21}+a_{23}*b_{31}+a_{20}*b_{01}$	$c_{22} = a_{22}*b_{22}+a_{23}*b_{32}+a_{20}*b_{02}+a_{21}*b_{12}$	$c_{23} = a_{23}*b_{33}+a_{20}*b_{03}+a_{21}*b_{13}+a_{22}*b_{23}$
Clock Cycle 12	a_{30}, b_{00}	a_{31}, b_{11}	a_{32}, b_{22}	a_{33}, b_{33}
Clock Cycle 13	a_{31}, b_{10}	a_{32}, b_{21}	a_{33}, b_{32}	a_{30}, b_{03}
Clock Cycle 14	a_{32}, b_{20}	a_{33}, b_{31}	a_{30}, b_{02}	a_{31}, b_{13}
Clock Cycle 15	a_{33}, b_{30}	a_{30}, b_{01}	a_{31}, b_{12}	a_{32}, b_{23}
Results	$c_{30} = a_{30}*b_{00}+a_{31}*b_{10}+a_{32}*b_{20}+a_{33}*b_{30}$	$c_{31} = a_{31}*b_{11}+a_{32}*b_{21}+a_{33}*b_{31}+a_{30}*b_{01}$	$c_{32} = a_{32}*b_{22}+a_{33}*b_{32}+a_{30}*b_{02}+a_{31}*b_{12}$	$c_{33} = a_{33}*b_{33}+a_{30}*b_{03}+a_{31}*b_{13}+a_{32}*b_{23}$

Table A2: Trident execution of a matrix product with the second matrix transposed ($C = A \times B^t$)

Input	MAC0	MAC1	MAC2	MAC3
Clock Cycle 0	a_{00}, b_{00}	a_{01}, b_{11}	a_{02}, b_{22}	a_{03}, b_{33}
Clock Cycle 1	a_{00}, b_{10}	a_{01}, b_{21}	a_{02}, b_{32}	a_{03}, b_{03}
Clock Cycle 2	a_{00}, b_{20}	a_{01}, b_{31}	a_{02}, b_{02}	a_{03}, b_{13}
Clock Cycle 3	a_{00}, b_{30}	a_{01}, b_{01}	a_{02}, b_{12}	a_{03}, b_{23}
Results	$c_{00} = a_{00}*b_{00}+a_{01}*b_{01}+a_{02}*b_{02}+a_{03}*b_{03}$	$c_{01} = a_{01}*b_{11}+a_{02}*b_{12}+a_{03}*b_{13}+a_{00}*b_{10}$	$c_{02} = a_{02}*b_{22}+a_{03}*b_{23}+a_{00}*b_{20}+a_{01}*b_{21}$	$c_{03} = a_{03}*b_{33}+a_{00}*b_{30}+a_{01}*b_{31}+a_{02}*b_{32}$
Clock Cycle 4	a_{10}, b_{00}	a_{11}, b_{11}	a_{12}, b_{22}	a_{13}, b_{33}
Clock Cycle 5	a_{10}, b_{10}	a_{11}, b_{21}	a_{12}, b_{32}	a_{13}, b_{03}
Clock Cycle 6	a_{10}, b_{20}	a_{11}, b_{31}	a_{12}, b_{02}	a_{13}, b_{13}
Clock Cycle 7	a_{10}, b_{30}	a_{11}, b_{01}	a_{12}, b_{12}	a_{13}, b_{23}
Results	$c_{10} = a_{10}*b_{00}+a_{11}*b_{01}+a_{12}*b_{02}+a_{13}*b_{03}$	$c_{11} = a_{11}*b_{11}+a_{12}*b_{12}+a_{13}*b_{13}+a_{10}*b_{10}$	$c_{12} = a_{12}*b_{22}+a_{13}*b_{23}+a_{10}*b_{20}+a_{11}*b_{21}$	$c_{13} = a_{13}*b_{33}+a_{10}*b_{30}+a_{11}*b_{31}+a_{12}*b_{32}$
Clock Cycle 8	a_{20}, b_{00}	a_{21}, b_{11}	a_{22}, b_{22}	a_{23}, b_{33}
Clock Cycle 9	a_{20}, b_{10}	a_{21}, b_{21}	a_{22}, b_{32}	a_{23}, b_{03}
Clock Cycle 10	a_{20}, b_{20}	a_{21}, b_{31}	a_{22}, b_{02}	a_{23}, b_{13}
Clock Cycle 11	a_{20}, b_{30}	a_{21}, b_{01}	a_{22}, b_{12}	a_{23}, b_{23}
Results	$c_{20} = a_{20}*b_{00}+a_{21}*b_{01}+a_{22}*b_{02}+a_{23}*b_{03}$	$c_{21} = a_{21}*b_{11}+a_{22}*b_{12}+a_{23}*b_{13}+a_{20}*b_{10}$	$c_{22} = a_{22}*b_{22}+a_{23}*b_{23}+a_{20}*b_{20}+a_{21}*b_{21}$	$c_{23} = a_{23}*b_{33}+a_{20}*b_{30}+a_{21}*b_{31}+a_{22}*b_{32}$
Clock Cycle 12	a_{30}, b_{00}	a_{31}, b_{11}	a_{32}, b_{22}	a_{33}, b_{33}
Clock Cycle 13	a_{30}, b_{10}	a_{31}, b_{21}	a_{32}, b_{32}	a_{33}, b_{03}
Clock Cycle 14	a_{30}, b_{20}	a_{31}, b_{31}	a_{32}, b_{02}	a_{33}, b_{13}
Clock Cycle 15	a_{30}, b_{30}	a_{31}, b_{01}	a_{32}, b_{12}	a_{33}, b_{23}
Results	$c_{30} = a_{30}*b_{00}+a_{31}*b_{01}+a_{32}*b_{02}+a_{33}*b_{03}$	$c_{31} = a_{31}*b_{11}+a_{32}*b_{12}+a_{33}*b_{13}+a_{30}*b_{10}$	$c_{32} = a_{32}*b_{22}+a_{33}*b_{23}+a_{30}*b_{20}+a_{31}*b_{21}$	$c_{33} = a_{33}*b_{33}+a_{30}*b_{30}+a_{31}*b_{31}+a_{32}*b_{32}$

Table A3: Trident execution of a matrix product with the first matrix transposed ($C = A^t \times B$)

Input	MAC0	MAC1	MAC2	MAC3
Clock Cycle 0	a_{00}, b_{00}	a_{01}, b_{01}	a_{02}, b_{02}	a_{03}, b_{03}
Clock Cycle 1	a_{01}, b_{00}	a_{02}, b_{01}	a_{03}, b_{02}	a_{00}, b_{03}
Clock Cycle 2	a_{02}, b_{00}	a_{03}, b_{01}	a_{00}, b_{02}	a_{01}, b_{03}
Clock Cycle 3	a_{03}, b_{00}	a_{00}, b_{01}	a_{01}, b_{02}	a_{02}, b_{03}
Results	$c_{00} = a_{00}*b_{00}$ $c_{10} = a_{01}*b_{00}$ $c_{20} = a_{02}*b_{00}$ $c_{30} = a_{03}*b_{00}$	$c_{11} = a_{01}*b_{01}$ $c_{21} = a_{02}*b_{01}$ $c_{31} = a_{03}*b_{01}$ $c_{01} = a_{00}*b_{01}$	$c_{22} = a_{02}*b_{02}$ $c_{32} = a_{03}*b_{02}$ $c_{02} = a_{00}*b_{02}$ $c_{12} = a_{01}*b_{02}$	$c_{33} = a_{03}*b_{03}$ $c_{03} = a_{00}*b_{03}$ $c_{13} = a_{01}*b_{03}$ $c_{23} = a_{02}*b_{03}$
Clock Cycle 4	a_{10}, b_{10}	a_{11}, b_{11}	a_{12}, b_{12}	a_{13}, b_{13}
Clock Cycle 5	a_{11}, b_{10}	a_{12}, b_{11}	a_{13}, b_{12}	a_{10}, b_{13}
Clock Cycle 6	a_{12}, b_{10}	a_{13}, b_{11}	a_{10}, b_{12}	a_{11}, b_{13}
Clock Cycle 7	a_{13}, b_{10}	a_{10}, b_{11}	a_{11}, b_{12}	a_{12}, b_{13}
Results	$c_{00} = a_{00}*b_{00} + a_{10}*b_{10}$ $c_{10} = a_{01}*b_{00} + a_{11}*b_{10}$ $c_{20} = a_{02}*b_{00} + a_{12}*b_{10}$ $c_{30} = a_{03}*b_{00} + a_{13}*b_{10}$	$c_{11} = a_{01}*b_{01} + a_{11}*b_{11}$ $c_{21} = a_{02}*b_{01} + a_{12}*b_{11}$ $c_{31} = a_{03}*b_{01} + a_{13}*b_{11}$ $c_{01} = a_{00}*b_{01} + a_{10}*b_{11}$	$c_{22} = a_{02}*b_{02} + a_{12}*b_{12}$ $c_{32} = a_{03}*b_{02} + a_{13}*b_{12}$ $c_{02} = a_{00}*b_{02} + a_{10}*b_{12}$ $c_{12} = a_{01}*b_{02} + a_{11}*b_{12}$	$c_{33} = a_{03}*b_{03} + a_{13}*b_{13}$ $c_{03} = a_{00}*b_{03} + a_{10}*b_{13}$ $c_{13} = a_{01}*b_{03} + a_{11}*b_{13}$ $c_{23} = a_{02}*b_{03} + a_{12}*b_{13}$
Clock Cycle 8	a_{20}, b_{20}	a_{21}, b_{21}	a_{22}, b_{22}	a_{23}, b_{23}
Clock Cycle 9	a_{21}, b_{20}	a_{22}, b_{21}	a_{23}, b_{22}	a_{20}, b_{23}
Clock Cycle 10	a_{22}, b_{20}	a_{23}, b_{21}	a_{20}, b_{22}	a_{21}, b_{23}
Clock Cycle 11	a_{23}, b_{20}	a_{20}, b_{21}	a_{21}, b_{22}	a_{22}, b_{23}
Results	$c_{00} = a_{00}*b_{00} + a_{10}*b_{10} + a_{20}*b_{20}$ $c_{10} = a_{01}*b_{00} + a_{11}*b_{10} + a_{21}*b_{20}$ $c_{20} = a_{02}*b_{00} + a_{12}*b_{10} + a_{22}*b_{20}$ $c_{30} = a_{03}*b_{00} + a_{13}*b_{10} + a_{23}*b_{20}$	$c_{11} = a_{01}*b_{01} + a_{11}*b_{11} + a_{21}*b_{21}$ $c_{21} = a_{02}*b_{01} + a_{12}*b_{11} + a_{22}*b_{21}$ $c_{31} = a_{03}*b_{01} + a_{13}*b_{11} + a_{23}*b_{21}$ $c_{01} = a_{00}*b_{01} + a_{10}*b_{11} + a_{20}*b_{21}$	$c_{22} = a_{02}*b_{02} + a_{12}*b_{12} + a_{22}*b_{22}$ $c_{32} = a_{03}*b_{02} + a_{13}*b_{12} + a_{23}*b_{22}$ $c_{02} = a_{00}*b_{02} + a_{10}*b_{12} + a_{20}*b_{22}$ $c_{12} = a_{01}*b_{02} + a_{11}*b_{12} + a_{21}*b_{22}$	$c_{33} = a_{03}*b_{03} + a_{13}*b_{13} + a_{23}*b_{23}$ $c_{03} = a_{00}*b_{03} + a_{10}*b_{13} + a_{20}*b_{23}$ $c_{13} = a_{01}*b_{03} + a_{11}*b_{13} + a_{21}*b_{23}$ $c_{23} = a_{02}*b_{03} + a_{12}*b_{13} + a_{22}*b_{23}$
Clock Cycle 12	a_{30}, b_{30}	a_{31}, b_{31}	a_{32}, b_{32}	a_{33}, b_{33}
Clock Cycle 13	a_{31}, b_{30}	a_{32}, b_{31}	a_{33}, b_{32}	a_{30}, b_{33}
Clock Cycle 14	a_{32}, b_{30}	a_{33}, b_{31}	a_{30}, b_{32}	a_{31}, b_{33}
Clock Cycle 15	a_{33}, b_{30}	a_{30}, b_{31}	a_{31}, b_{32}	a_{32}, b_{33}
Results	$c_{00} = a_{00}*b_{00} + a_{10}*b_{10} + a_{20}*b_{20} + a_{30}*b_{30}$ $c_{10} = a_{01}*b_{00} + a_{11}*b_{10} + a_{21}*b_{20} + a_{31}*b_{30}$ $c_{20} = a_{02}*b_{00} + a_{12}*b_{10} + a_{22}*b_{20} + a_{32}*b_{30}$ $c_{30} = a_{03}*b_{00} + a_{13}*b_{10} + a_{23}*b_{20} + a_{33}*b_{30}$	$c_{11} = a_{01}*b_{01} + a_{11}*b_{11} + a_{21}*b_{21} + a_{31}*b_{31}$ $c_{21} = a_{02}*b_{01} + a_{12}*b_{11} + a_{22}*b_{21} + a_{32}*b_{31}$ $c_{31} = a_{03}*b_{01} + a_{13}*b_{11} + a_{23}*b_{21} + a_{33}*b_{31}$ $c_{01} = a_{00}*b_{01} + a_{10}*b_{11} + a_{20}*b_{21} + a_{30}*b_{31}$	$c_{22} = a_{02}*b_{02} + a_{12}*b_{12} + a_{22}*b_{22} + a_{32}*b_{32}$ $c_{32} = a_{03}*b_{02} + a_{13}*b_{12} + a_{23}*b_{22} + a_{33}*b_{32}$ $c_{02} = a_{00}*b_{02} + a_{10}*b_{12} + a_{20}*b_{22} + a_{30}*b_{32}$ $c_{12} = a_{01}*b_{02} + a_{11}*b_{12} + a_{21}*b_{22} + a_{31}*b_{32}$	$c_{33} = a_{03}*b_{03} + a_{13}*b_{13} + a_{23}*b_{23} + a_{33}*b_{33}$ $c_{03} = a_{00}*b_{03} + a_{10}*b_{13} + a_{20}*b_{23} + a_{30}*b_{33}$ $c_{13} = a_{01}*b_{03} + a_{11}*b_{13} + a_{21}*b_{23} + a_{31}*b_{33}$ $c_{23} = a_{02}*b_{03} + a_{12}*b_{13} + a_{22}*b_{23} + a_{32}*b_{33}$

TableA4: Trident execution of a matrix addition ($C = A+B$)

Input	Add0	Add1	Add2	Add3
Clock Cycle 0	$c_{00}=a_{00}, b_{00}$	$c_{01}=a_{01}, b_{01}$	$c_{02}=a_{02}, b_{02}$	$c_{03}=a_{03}, b_{03}$
Clock Cycle 1	$c_{10}=a_{10}, b_{10}$	$c_{11}=a_{11}, b_{11}$	$c_{12}=a_{12}, b_{12}$	$c_{13}=a_{13}, b_{13}$
Clock Cycle 2	$c_{20}=a_{20}, b_{20}$	$c_{21}=a_{21}, b_{21}$	$c_{22}=a_{22}, b_{22}$	$c_{23}=a_{23}, b_{23}$
Clock Cycle 3	$c_{30}=a_{30}, b_{30}$	$c_{31}=a_{31}, b_{31}$	$c_{32}=a_{32}, b_{32}$	$c_{33}=a_{33}, b_{33}$